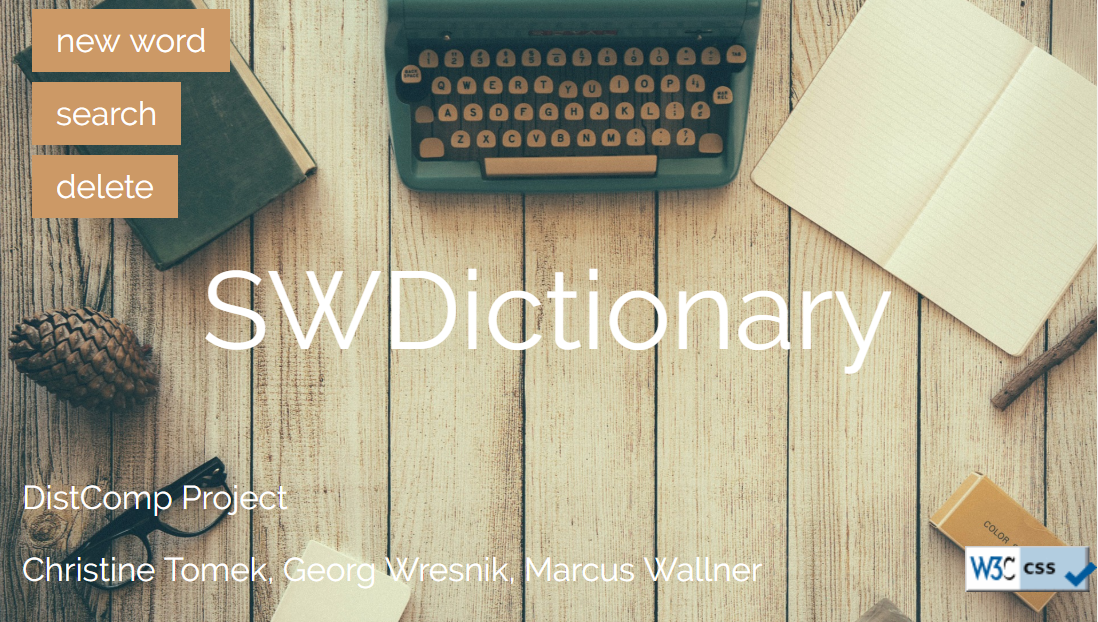
SWDICTIONARY

EIN FREIES COMMUNITY-WÖRTERBUCH



DISTRIBUTED COMPUTING

SWD ’16 – SS 2018

TOMEK, WALLNER, WRESNIK

1. CONTENT / Inhalt

[2](#_gjdgxs) Einleitung 3

[2.1](#_30j0zll) Ausgangssituation 3

[2.2](#_1fob9te) Projektübersicht / Übersicht 3

[3](#_3znysh7) Aufgabenstellung 3

[3.1](#_2et92p0) Client UI 3

[3.2](#_tyjcwt) Messaging Queuing / Messaging Queue(s) 3

[3.3](#_3dy6vkm) Worker / Consumer 3

[3.4](#_1t3h5sf) Web Socket Communication 4

[4](#_4d34og8) Setup / Installation 4

[4.1](#_2s8eyo1) Wie bekommt man das Projekt zum Laufen 4

[5](#_17dp8vu) Anleitung / BEdienung 4

[5.1](#_3rdcrjn) Client UI 4

[5.2](#_26in1rg) Messaging Queuing / Messaging Queue(s) 4

[5.3](#_lnxbz9) Worker / Consumer 5

[5.4](#_35nkun2) Web Socket Communication 5

[6](#_1ksv4uv) Summary / Zusammenfassung / Resüme 5

[6.1](#_44sinio) Output / Ergebnis 5

[6.2](#_2jxsxqh) Known Problems / Bekannte Probleme 5

[6.3](#_z337ya) Outlook / Ausblick / weitere Möglichkeiten 5

[6.4](#_3j2qqm3) Opinion / Persönliche Meinung / Erkenntnisse 5

[7](#_1y810tw) Anhang 5

[7.1](#_4i7ojhp) Code Repository / Sources 5

[7.2](#_2xcytpi) Links 5

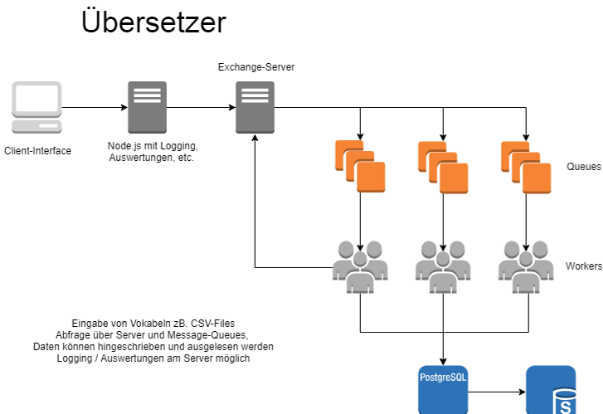
# Einleitung

## Ausgangssituation

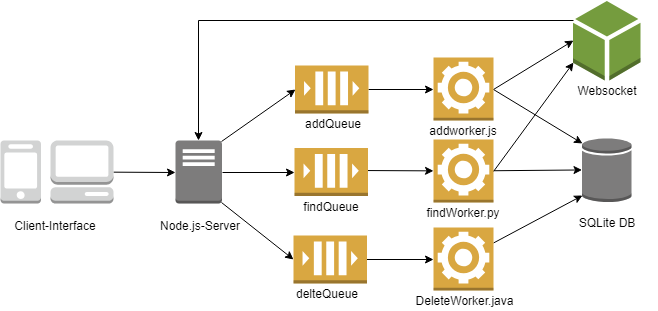
Für das Studienfach Distributed Computing bei Hr. Michael Ulm soll eine Projektarbeit erstellt werden, die den Semesterstoff des 4. Semesters des Studiengangs „Software Design (berufsbegleitende Vertiefung von Internettechnik“) umfasst und das Gelernte bei den Studenten praktisch vertieft.

## Projektübersicht / Übersicht

Erstentwurf von Mai 2018:



Endversion vom Juli 2018:



# Aufgabenstellung

## Client UI

–zB. Web Application mit Ein- & Ausgabe

## Messaging Queuing / Messaging Queue(s)

–Producer & Consumers

## Worker / Consumer

–Mindestens 3 verschiedene Programmiersprachen, zB. Python, Java, PHP

– Mindestens 2 verschiedene Worker, zB. bruteforce, dictionary

– Mindestens 2 verschiedene Daten (Quellen), zB. Datenbank, Dateisystem Logging

## Web Socket Communication

–Mehrfache Verbindungen von User Interface oder Workers

# Setup / Installation

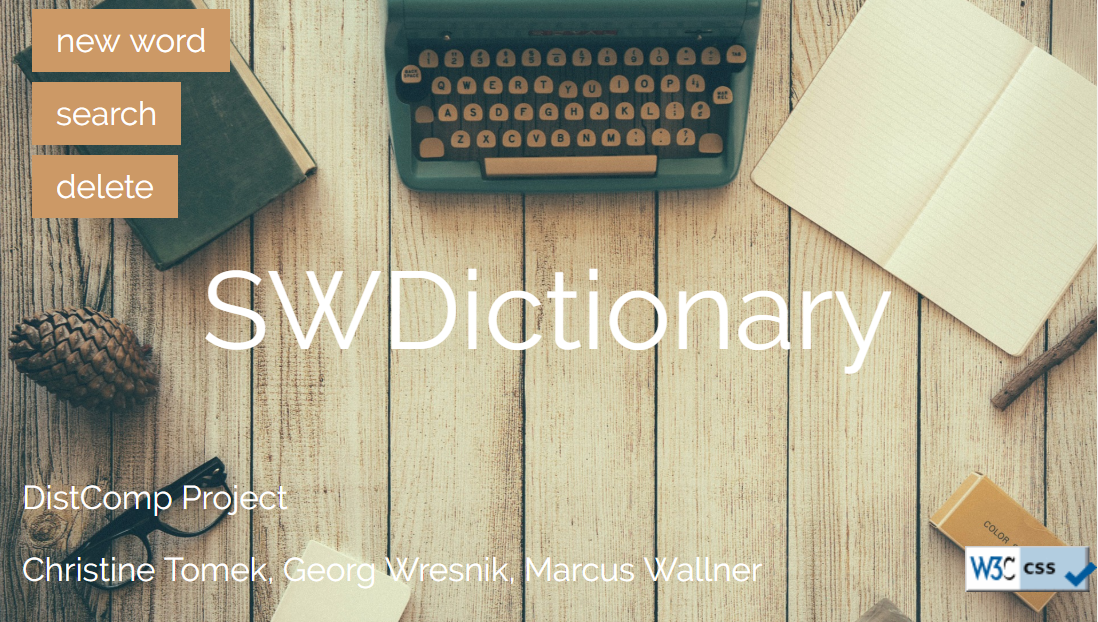
## Wie man das Projekt zum Laufen bekommt

* Kontrolliere durch Eingabe in der Kommandozeile (cmd.exe) ob und welche Version von node.js installiert ist:  
  where node  
  node --version  
  Wenn die Version unter v7.6 ist muss sie aktualisiert werden.  
  Entweder benutze die aktuellste Version von [node.js](https://nodejs.org/en/download/) oder folge diesen Schritten: [update node version - on Windows](https://github.com/wresnikg16/Translate_DistComp/blob/master/documentation/updateNodeVersion.md).
* Downloade und installiere [RabbitMQ](https://www.rabbitmq.com/download.html) und die notwendigen [Plugins](https://www.rabbitmq.com/management.html).
* Kontrolliere ob die Umgebungsvariable die Pfade zu den executables für Node, Python und Java beinhaltet.
* Tippe .\start.bat in die Kommandozeile um
  + die notwendigen node Module zu installieren,
  + Python Bibliotheken zu installieren,
  + die drei Worker zu starten,
  + und den node Webserver zu starten.
* Öffne den Browser auf [http://localhost:8081](http://localhost:8081/)
* Im add/search/delete Bereich der Webseite können Wörter hinzugefügt, gesucht und aus dem Wörterbuch gelöscht werden.

# Anleitung / BEdienung

## Client UI

Das GUI (graphical user interface) der vorgestellten Web-Applikation ist schlicht und möglichst selbsterklären aufgebaut. Im linken oberen Bereich befindet sich die Navigation zu den angebotenen Ressourcen. Klickt man auf einen der Buttons so wird man auf die Subseiten /sites/add.html, /sites/search.html und /sites/remove.html weitergeleitet, die die jeweilige Funktionalität des Wörterbuchs anbieten.



## Frontend Webserver server.js

Der node.js-Server server.js stellt im Projekt den Frontend Webserver dar. Dieser stellt dem User die Benutzeroberfläche zur Verfügung. Der Server nutzt folgende Struktur, welche anschließend kurz beschrieben wird.

Unter dem Ordner „sites“ sind die einzelnen HTML Files zu finden. Für jede einzelne Funktion gibt es eine eigene HTML Seite und natürlich die „index.html“ als Startseite. Die Anforderung jeder Seite wird vom NodeJS Server übernommen, über die entsprechende URL wird die gewünschte Seite an den Client ausgegeben.

Auch das style.css-File, das Background-Image und das scripts.js-File für sonstige kleinere Funktionen finden sich am node.js-Server.

Abbildung 1 node.js Server Files

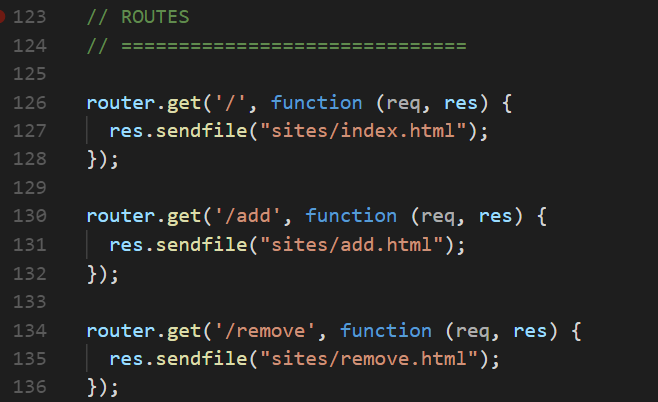


Abbildung 2 server.js routing

Für das Ausführen der Funktionen Add, Find und Delete gibt es ebenfalls ein eigenes routing. Sobald man die Url aufruft werden die Informationen an die Queues geschickt, wo sie anschließend abgearbeitet werden von den jeweiligen worker.

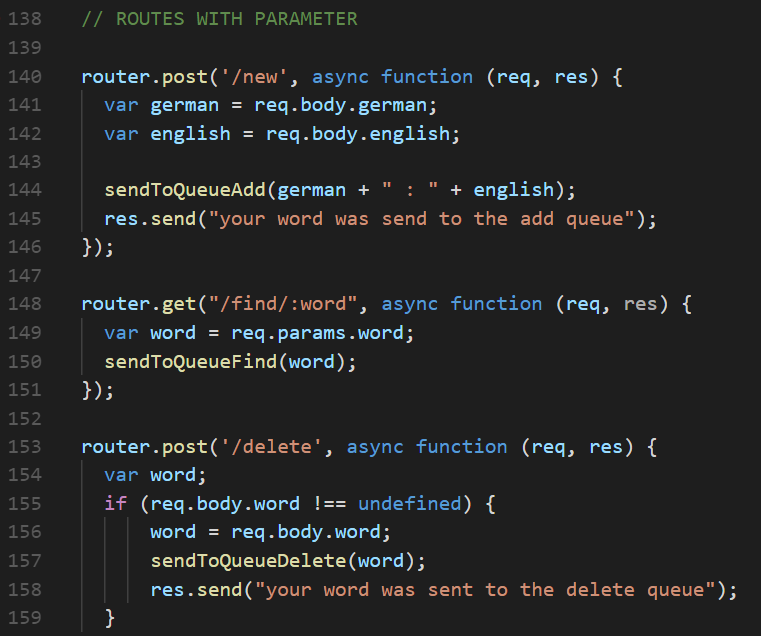


Abbildung 3 server.js function routing

Der NodeJS Server selbst wird ganz zum Schluss gestartet (Port 8081), dieser liest auch die Konfiguration aus der „config.yml“ Datei aus, was die Möglichkeit gibt die Server Einstellungen ohne Code Anpassung zu verändern.



Abbildung 4 server.js server start

## Messaging Queuing / Messaging Queue(s)

Der Message Broker RabbitMQ wurde verwendet um die Funktionalität von Message Queues zu implementieren. Er unterstützt zahlreiche verschiedene Programmiersprache und bietet auch ein Web-Management-Tool an, das unter <http://localhost:15672/> mittels Username: „guest“ und Passwort: „guest“ aufgerufen werden kann. Jede Ressource bzw. die dort ausgeführte Javascript Methode schreibt Daten in eine eigene Queue. Diese Queue buffert die Daten, bis sie vom jeweils zuständigen Worker abgeholt und verarbeitet werden.

Durch die Einbindung der Library „amqplib/callback\_api“ im NodeJS Server ist es möglich Nachrichten an die Queue zu übermitteln. Es wird für jede Funktion eine eigene Queue verwendet die in der Funktion definiert wird.

* Find 🡪 findQueue
* Delete 🡪 deleteQueue
* Add 🡪 addQueue

Sollte es die Queue nicht geben wird diese vom Server angelegt.

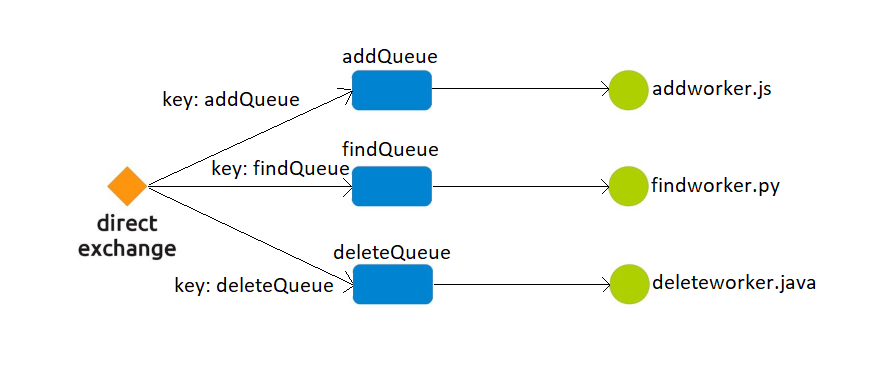
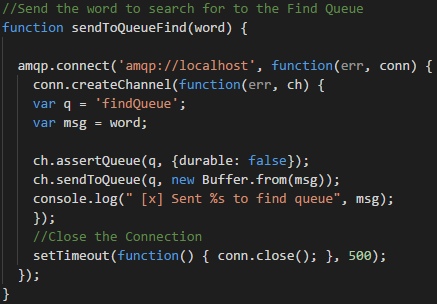


Abbildung 5 Queue Grafik



Dieses Code Snippet zeigt das Senden einer Nachricht an die „findQueue“ mittels AMQP, dem Advanced Message Queuing Protocol“, dass von RabbitMQ verwendet wird.

Abbildung 6 Beispiel send to Queue

## Web Socket Communication

Um von den Workern eine Rückmeldung zu bekommen, wird ebenfalls mit dem Webserver ein Websocket Server auf Port 8082 gestartet. Die Konfiguration wird dabei gleich am Anfang im server.js vorgenommen.

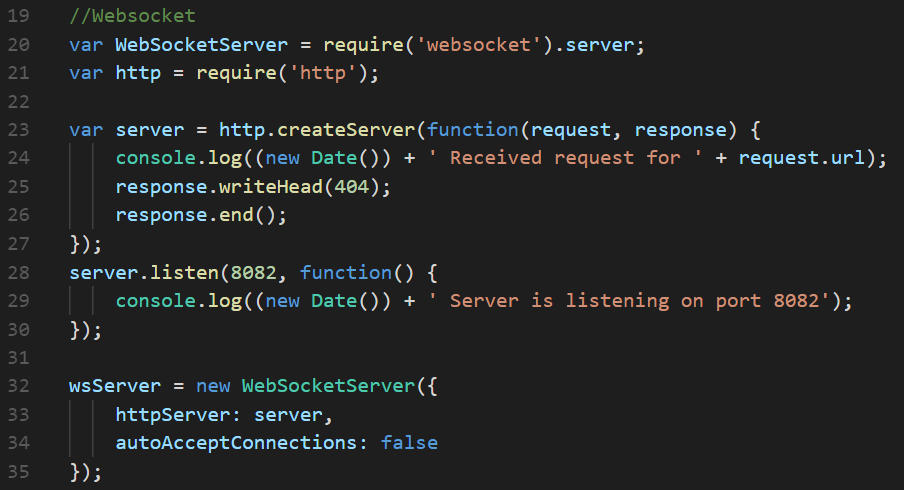


Abbildung 7 Websocket Server Konfiguration

Die EventHandler für den Websocket Server sind wie folgt definiert, die Rückmeldung vom Worker wird in der Konsole ausgegeben und dem Worker wird zurückgemeldet, dass seine Meldung angekommen ist.

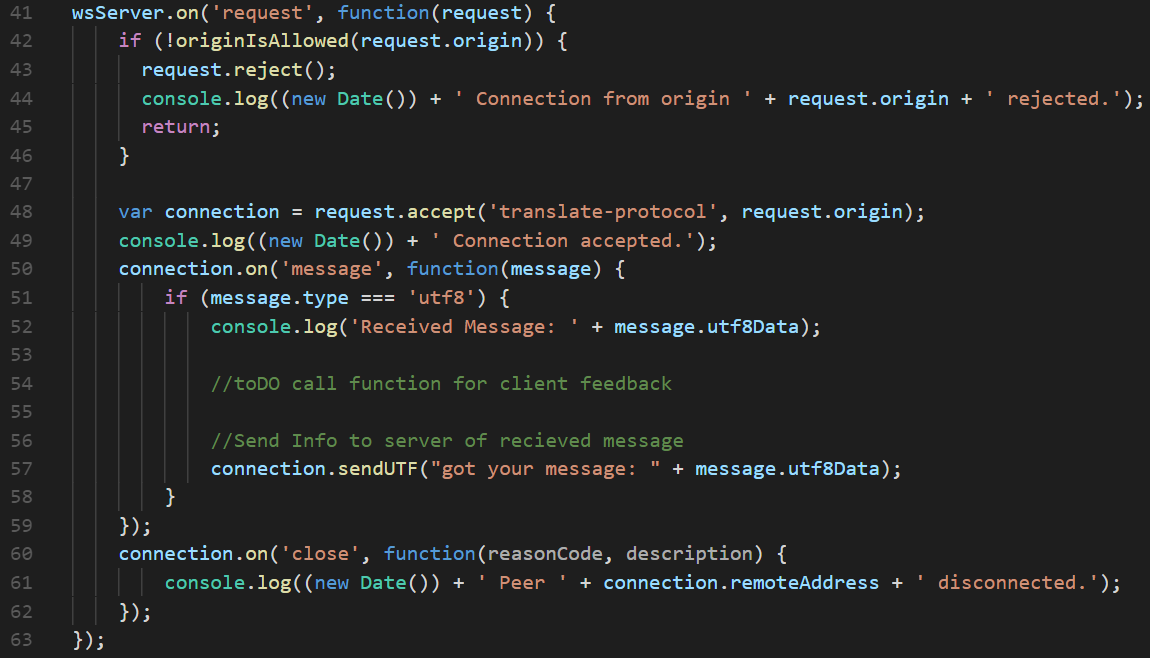


Abbildung 8 Websocket Eventhandler

## Worker / Consumer

Um die drei angebotenen Funktionalitäten (hinzufügen, suchen und löschen) auszuführen, wird jede Queue von einem spezifischen Worker abgefragt und die Daten daraus verarbeitet.

Es wurden 3 Worker erstellt um die Funktionen Find, Delete und Add abzubilden. Die Add Funktion wird von einem Javascript Worker erledigt, die Find von einem Python Worker und die Delete Funktion von einem Java Worker.

## addworker.js

Dieser Worker ist für das Hinzufügen von Daten in die Datenbank zuständig. Dieser Worker erstellt bei erstmaliger Ausführung mit der Funktion „init()“ die Datenbank und legt auch die benötigte Tabelle an.

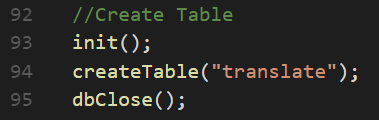


Abbildung 9 Create DB

Der Hauptteil des Workers ist das er ständig alle Messages aus der „addQueue“ holt und anschließend die Daten in die die Datenbank schreibt, dies macht er mittels entsprechendem Insert Statement. Ebenfalls hat er die Rückmeldung mittels Websocket implementiert. Folgendes Code Snippet zeigt das Abfragen aus der Queue und die Verarbeitungsschritte bis zum Rückmelden an des Websocket Server.

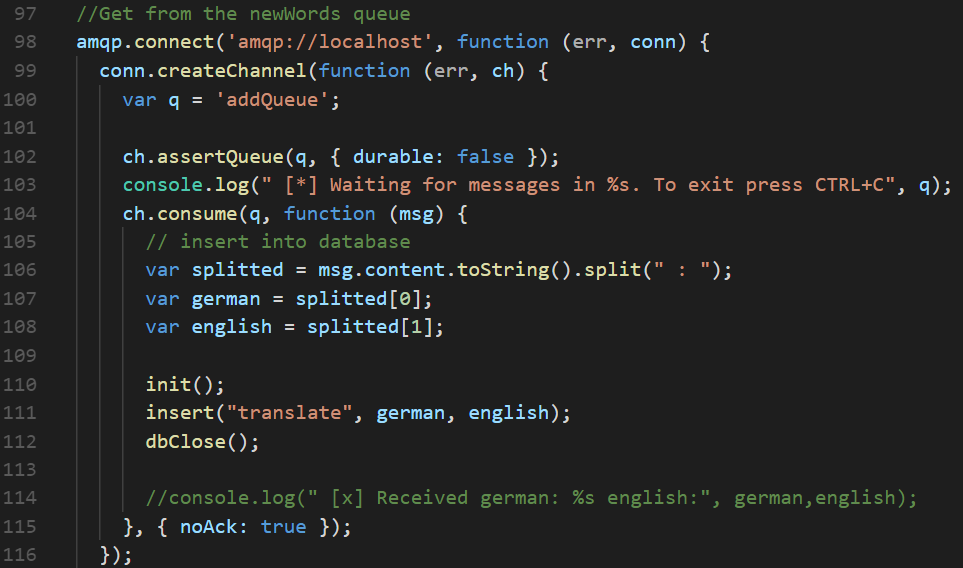


Abbildung 10 Consume addworker.js

Die Rückmeldung an den Websocket geschieht in der „insert()“ Funktion, dort wird je nach Erfolg oder Misserfolg beim Einfügen die entsprechende Nachricht an den Server gesendet.



Abbildung 11 insert and websocket message

Es folgt die Websocket Konfiguration von addworker.js, dabei ist es vor allem wichtig das über „client.connect()“ das richtige Protokoll angeben wird, wir haben das passend für unser Beispiel „translate-protocol“ genannt. Der Server weiß sonst nicht wie er damit umgehen soll, wenn er etwas nicht definiertes bekommt.



Abbildung 12 addworker.js Websocket config

## findWorker.py

Dieser Worker ist in Python implementiert und für die Suche nach Einträgen in der Datenbank zuständig. Der Begriff, der auf der search Site eingegeben wird, wird an die „findQueue“ gesendet und von diesem Worker abgeholt und bearbeitet. Folgender Code beschreibt die Verbindung zur Queue:

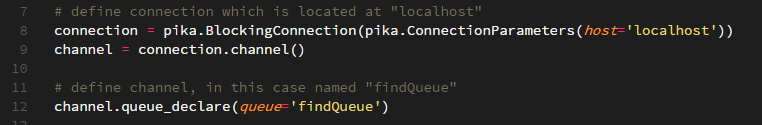


Abbildung Verbindungsaufbau zur findQueue

In der nächsten Abbildung ist ersichtlich wie die Verbindung zur Datenbank hergestellt wird. Die Funktion „get\_entry“ stellt über den „with conn“ Block eine Verbindung durch den zuvor angegebenen Parameter zur Datenbank her und schließt diese beim Verlassen des Blockes wieder. Über den Cursor wird mit „execute“ nach Einträgen in der Datenbank gesucht. Es ist in dieser Implementierung unerheblich ob vom User das deutsche oder das englische Wort gesucht wird. Wenn ein Eintrag gefunden wird, werden jeweils der deutsche und der englische Begriff an den Server gesendet.

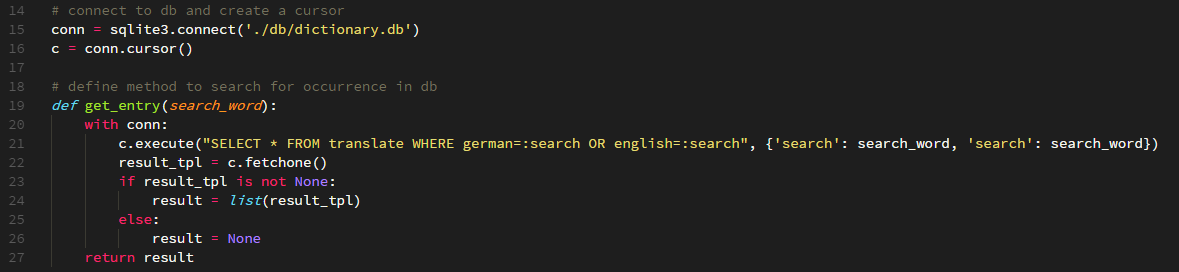


Abbildung Herstellung der Datenbankverbindung und Definition der Such Funktion

Der folgende Code in Abbildung 15 beschreibt die Callback Funktion die ausgeführt wird, wenn eine Nachricht aus der „findQueue“ abgeholt wird. Der Suchbegriff wird decodiert und als Parameter an die „get\_entry“ Funktion übergeben. Die Begriffe, die hierbei gefunden werden, werden als List in der Result Variable gespeichert. Anschließend wird eine Verbindung zum Websocket des Servers hergestellt und über diese Verbindung das gefundene Wort an den Webserver übermittelt.

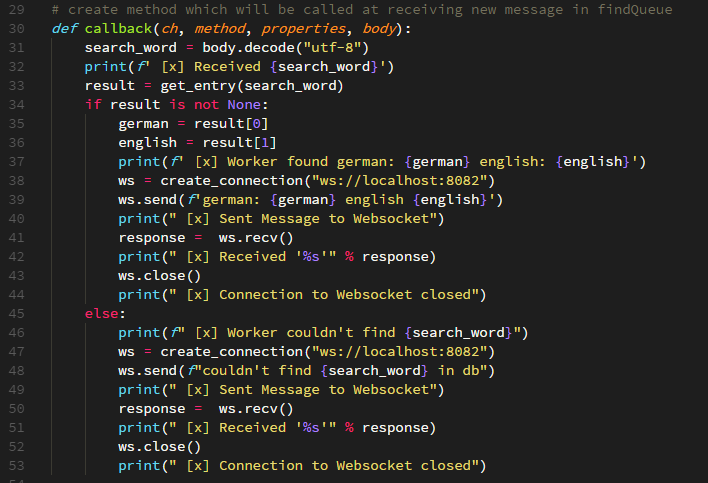


Abbildung Callback Funktion für eingehende Nachrichten in findQueue

## DeleteWorker.jar

Dieser Worker ist in Java geschrieben. Er holt die Daten aus der „deleteQueue“ und führt ein DELETE-Statement auf die SQLite Datenbank aus.

In der Main-Methode wird zuerst die Verbindung zur gewünschten Queue hergestellt. Danach wird in der handleDelivery() Methode zuerst das Wort aus der Datenbank gelöscht und danach die Queue geleert.

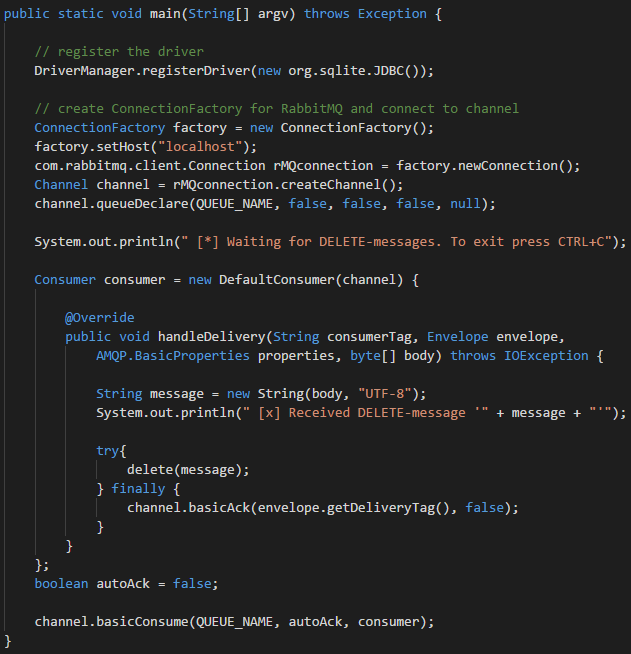


Abbildung 16 Java Main-Methode

In der delete() Methode selbst wurde aus Sicherheitsgründen mit PreparedStatements gearbeitet um SQL-Injections zu verhindern. Da die executeUpdate() Methode einen Wert größer 0 zurückgibt, falls ein entsprechendes Wort gelöscht werden konnte, wird über diesen Wert die Konsolen-Ausgabe gesteuert. Zum Schluss wird die offene Verbindung natürlich wieder geschlossen.

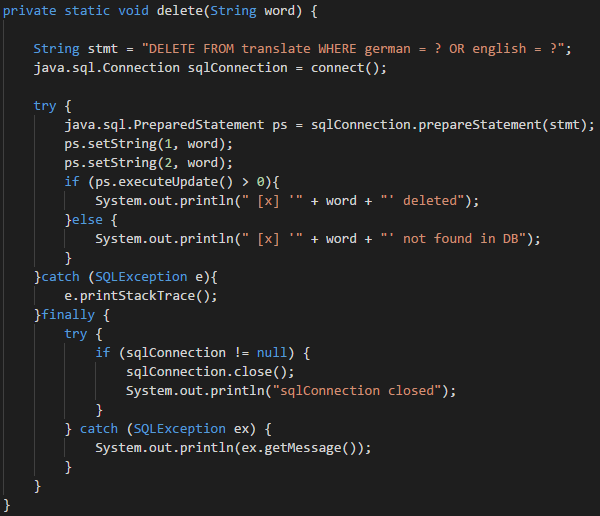


Abbildung 17 delete-Methode

Die connect() Methode stellt die Verbindung zur SQLite Datenbank her um den Löschvorgang darin zu ermöglichen. Wurde die Verbindung erfolgreich hergestellt, so wird diese Information in der Konsole ausgegeben.

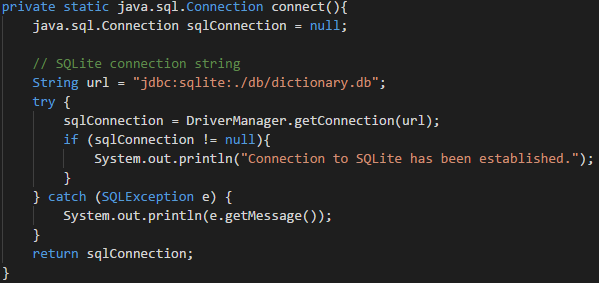


Abbildung 18 Verbindung zur SQLite Datenbank

# Zusammenfassung / Resüme

## Ergebnis

Um eine wirkliche sinnvolle Nutzung aller angeforderten technischen Möglichkeiten zu implementieren, wäre es sinnvoll, das Projekt nochmals vollständig zu überarbeiten. Zum aktuellen Zeitpunkt wurden die Technologien verwendet um den Mindestanforderungen zu entsprechen, jedoch wird teilweise nicht ihr volles Potenzial ausgeschöpft.

## Bekannte Probleme

### No Response zum Frontend

Leider war es uns nicht möglich die Rückmeldung von den Workern über Websocket am Client anzuzeigen, alternativ wurde eine Ausgabe in der jeweiligen Konsole implementiert.

### JAVA VERSION

Sollte es Probleme beim Ausführen des DeleteWorkers geben, kann die JavaVersion kontrolliert werden. Das .jar wurde mit Java Version 8. erstellt.

java version "1.8.0\_181"

Java(TM) SE Runtime Environment (build 1.8.0\_181-b13)

Java HotSpot(TM) Client VM (build 25.181-b13, mixed mode, sharing)

## Ausblick / weitere Möglichkeiten

Das Projekt könnte im Bezug auf klare Rückmeldungen an den User in der GUI noch erheblich verbessert werden, vor allem auch nach Abarbeitung der einzelnen Funktionalitäten.

Weiters könnten zusätzliche Checks der Eingabedaten zu höherer Sicherheit führen.

## Persönliche Meinung / Erkenntnisse

# Anhang

## Code Repository / Sources

GitRepo:

<https://github.com/wresnikg16/Translate_DistComp.git>

## Links

<https://www.rabbitmq.com/>